Signals, signals, mysterious signals…

Operating Systems, Fall 2026

Instructors: Travis McGaha

Head TAs: Shriya Sane

mosi-mosi?

Cool J-pop(?) song that talks about signals (not OS signals tho ☹)

Arnuv Batra
Jeffrey Oduman
Kevin Xue
Leo Zhang
Khoi Dinh
Abdullah Goher
Shreya Mukunthan
Sydney Fitzgerald
Monisha Krothapalli
Daniel Price
Rania Souissi
Veer Kakar
Wesley Liu

Poll Everywhere

pollev.com/tqm

- How is/was penn-vector?

Administrivia

- First Assignment (penn-vector)
  - Already posted
  - **"Due" this Friday, 9/04**
  - Mostly a C refresher

- Pre-Semester Survey
  - Anonymous
  - Short!
  - Already posted
  - **Due Wednesday, 9/09**

Administrivia

- Second Assignment (penn-shell)
  - Released on Tuesday
  - Partner assignment – start looking! Thread posted on Ed & google form to assist
  - Milestone 0 due Friday 9/11
  - Milestone 1 due Friday 9/18
  - Full assignment due Wednesday 9/23
  - Basic shell implementation
    - System calls, processes, I/O redirection, pipes, job control, and more!
  - Autograder posted sometime this week

- Check the course calendar for office hour times!

Lecture Outline

- **Software signals (resumed from Tuesday)**
- Signal blocking & Signal sets
- Busy waiting, sigsuspend & sigwait
- Critical sections & signal safety
- wait(), waitpid(), WNOHANG

Signals

- A Process can be interrupted with various types of signals
  - This interruption can occur in the middle of most code
- Each signal type has a different meaning, number associated with it, and a way it is handled
- These are different from an interrupt, but similar idea
  - signals are “higher level” and apply to a process. The kernel / some process will deliver the signal.
  - Interrupts are lower level mechanisms that cause the hardware to poke the kernel and respond
  - Some interrupts lead to a signal being sent (CTRL + C on keyboard -> SIGINT)

Signals

- A Process can be interrupted with various types of signals
  - This interruption can occur in the middle of most code
- Each signal type has a different meaning, number associated with it, and a way it is handled (disposition)

- Examples:
  - `SIGCHLD` → Default Disposition: ignore
  - `SIGINT`
  - `SIGKILL` → Default Disposition: terminate the process
  - `SIGALRM`
  - `SIGSEGV` → Default Disposition : terminate & core dump

Review: Changing what a Signal Does

```
int sigaction(int signum,
              struct sigaction* act,
              struct sigaction* old);
```

- `int signum`
  - The signals who'd disposition (behavior) is being changed.
- `struct sigaction* act`
  - The struct containing the function that will be called upon the delivery of a signal + other flags.
- `struct sigaction* old`
  - The struct containing the old function that would be called upon the delivery of a signal + other flags.
- You can not change the disposition of **`SIG_KILL`** and **`SIG_STOP`**.

Review: struct sigaction

```
struct sigaction {
  void     (*sa_handler)(int);
  int        sa_flags;
  ...
};
```

- Set sa_handler equal to the function (signal handler) we want to use
  - Set **`sa_handler`** to **`SIG_IGN`** to set disposition to IGNORE
  - Set **`sa_handler`** to **`SIG_DFL`** for default disposition

- In this class: set **`sa_flags`** to **`SA_RESTART`**
  - This makes it so that *certain* system calls are automatically restart/continue if they are interrupted by a signal. (wanna see the list? `man 7 signal` ☺)

alarm()

```
unsigned int alarm(unsigned int seconds);
```

- Delivers the **SIGALRM** signal to the calling process after the specified number of seconds
- Default **SIGALRM** disposition: ***terminate the process***
- How to cancel alarms?
  - I leave this as an exercise for you: try reading the man pages

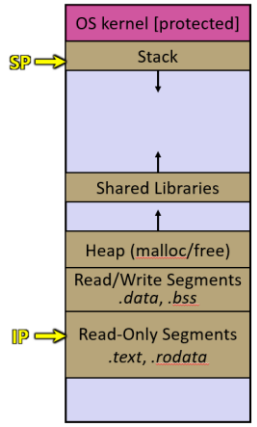
Demo no_sleep.c

- See `no_sleep.c`
  - "Sleeps" for 10 seconds without sleeping, using alarm
  - Brief code demo to see how to use a signal handler & alarm
  - Signal handler manipulates global state

Diagram: signals

User Processes

`./example`
`process id: 100`



OS

Process Table

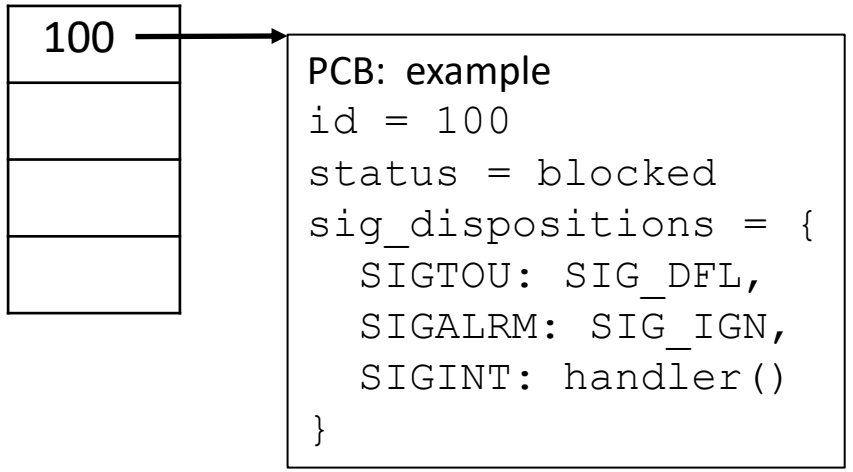


Diagram: signals

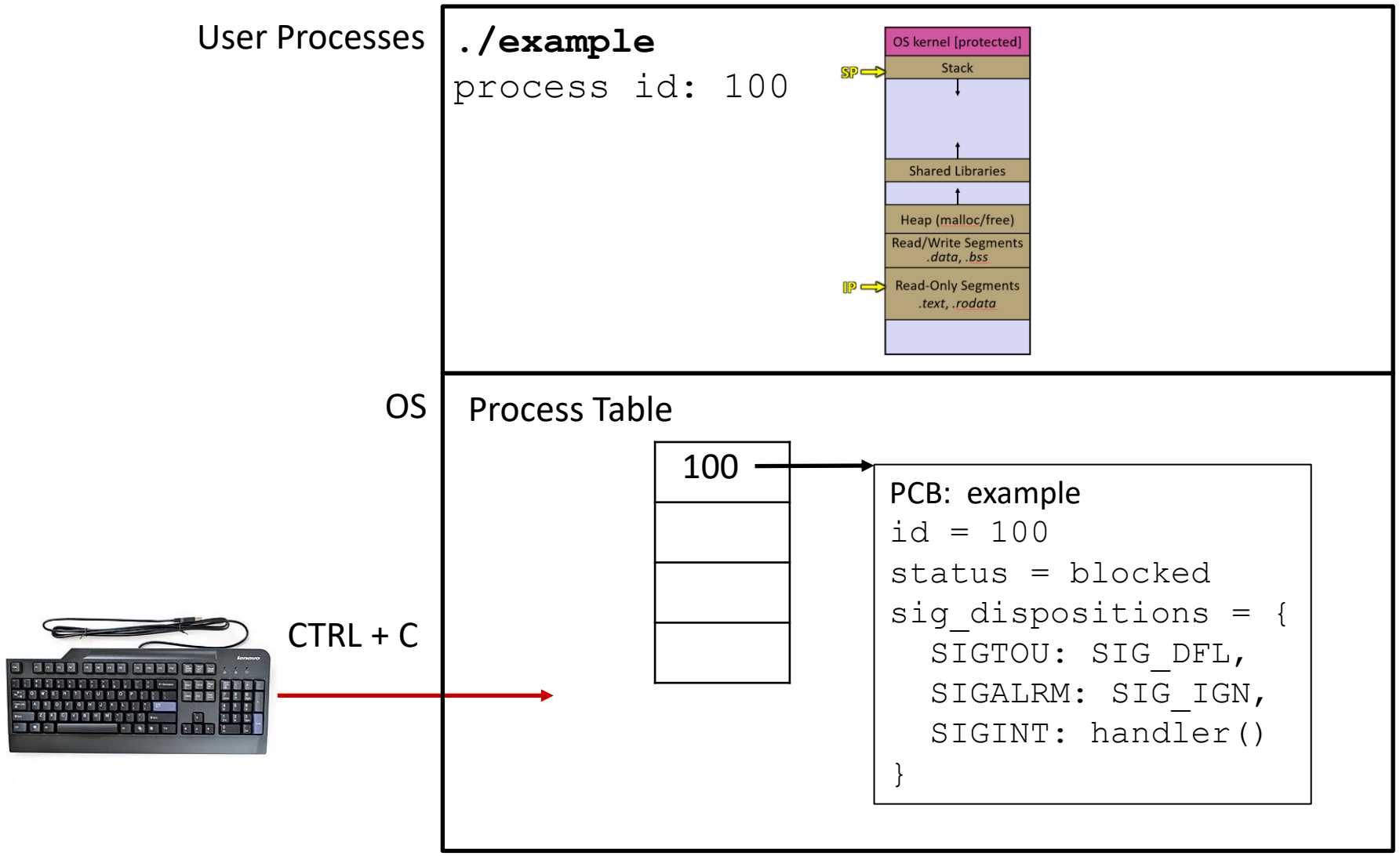
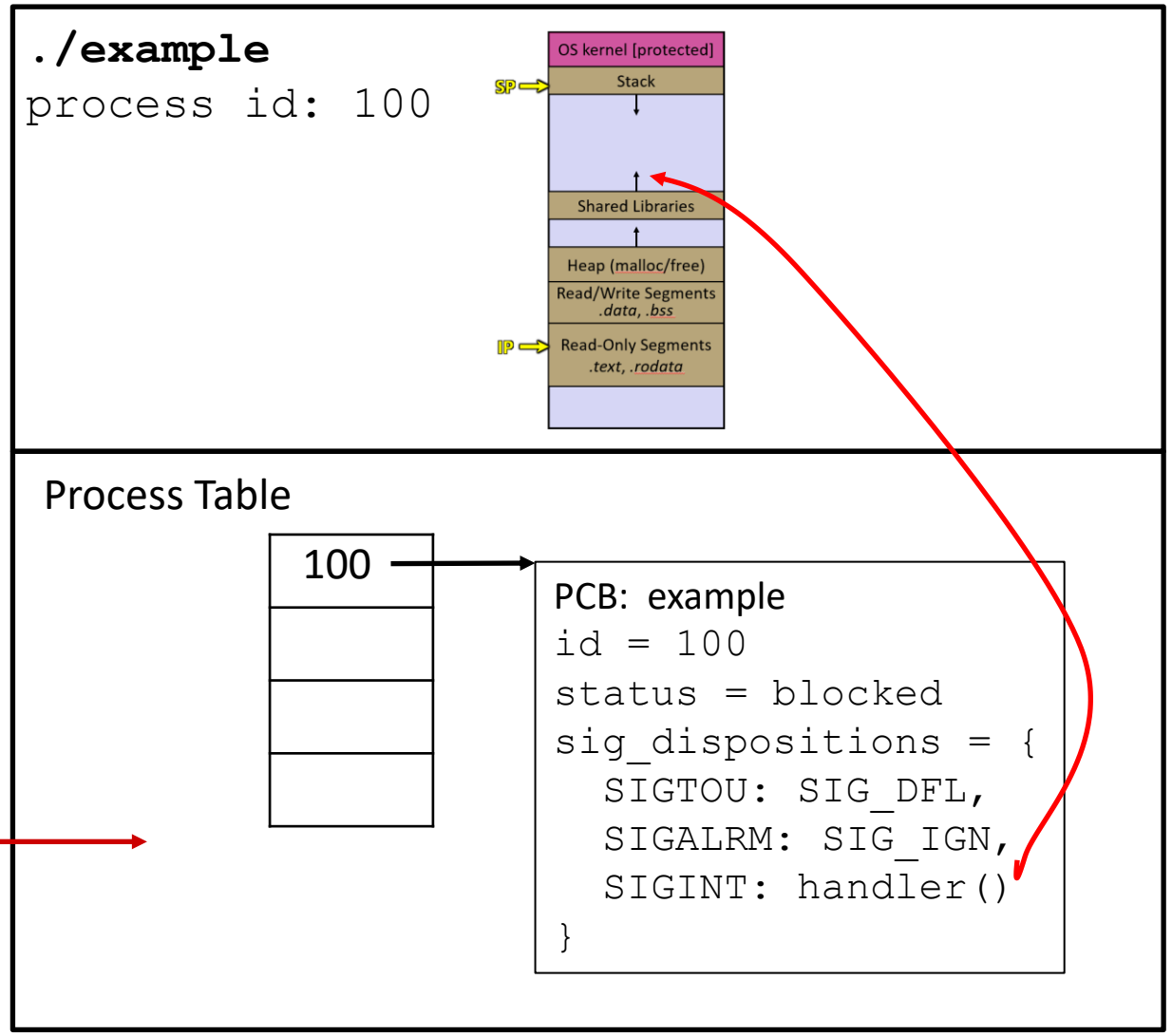


Diagram: signals

User Processes

Once a signal is received, the OS forwards the signal to the corresponding process.

CTRL + C



signal dispositions

- Every signal has a current *disposition*
  - This determines how the process behaves when it is delivered the signal from the OS.
- `Term`
  - Default action is to terminate the process.
- `Ign`
  - Default action is to ignore the signal.
- `Core`
  - Default action is to terminate the process and dump core (see core(5)).
- `Stop`
  - Default action is to stop the process.
- `Cont`
  - Default action is to continue the process if it is currently stopped.

And, as we've seen,
you can install your own signal handler;
a user defined 'disposition'.

SIGCHLD handler

- When child process is terminated or stopped, a **`SIGCHLD`** signal is received by the parent, and by default ignored.
  - It's `'disposition'` is `Ign`
- You can install a custom signal handler for **`SIGCHLD`**, and use that to help handle children update statuses:
  - This allows the parent process to do other things instead of blocking via **`wait()`** or **`waitpid()`**
  - **You might expect to receive a `SIGCHLD` anyways, so why waste time calling waitpid?**
  - **You could just call `waitpid()` when you need to…within a signal handler itself.**
  - **Or, set a flag to know to call it later. ☺**

- Relevant for proj2: **`penn-shell`**

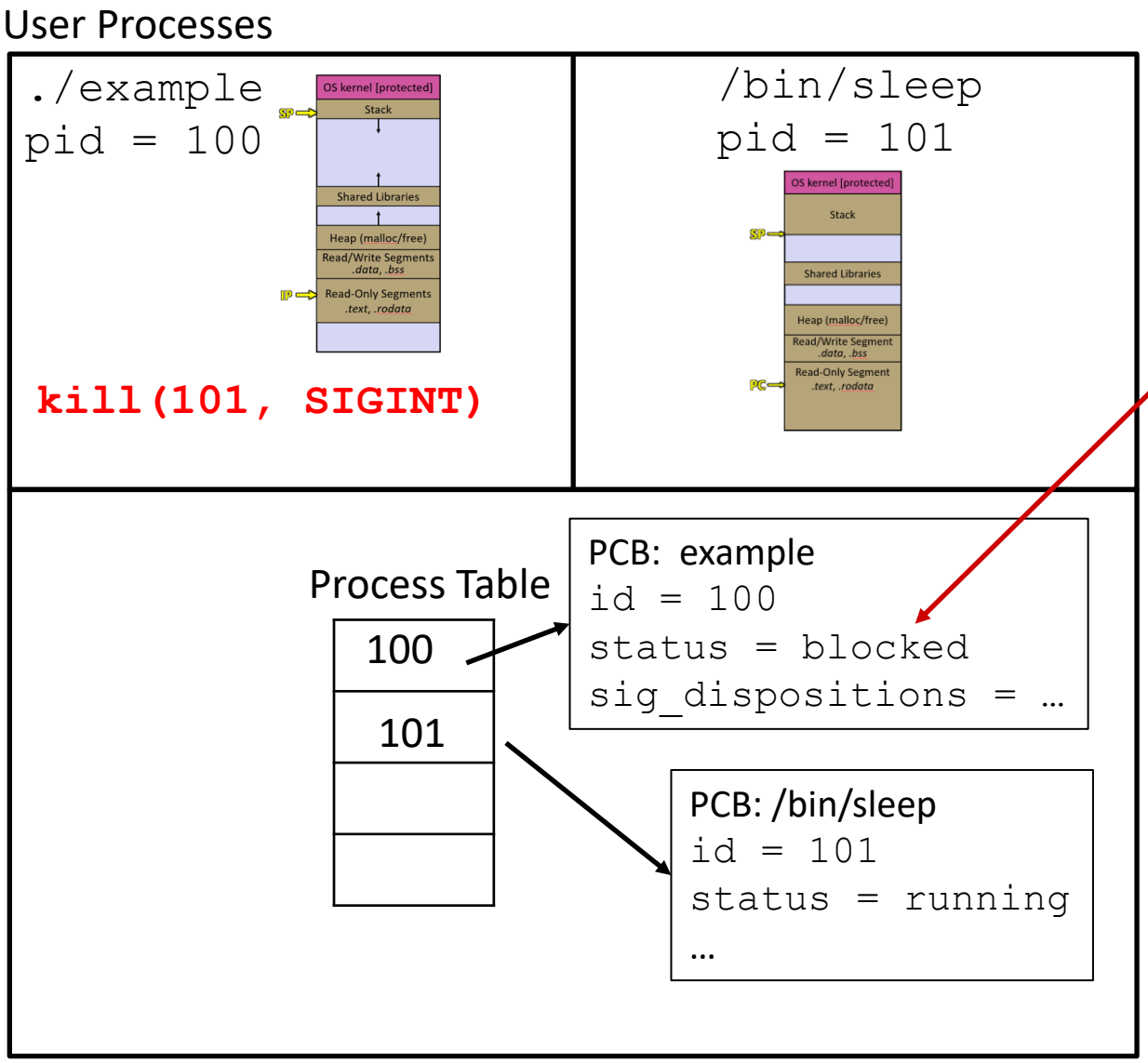
Review: `kill()`

- Allows us to send specific signals to a specific process.
- `int kill(pid_t pid, int sig);`
- `pid`: specifies the process
- `sig`: specifies the signal

- Example: `kill(child, SIGKILL);`
  - Delivers a `SIGKILL` to the process with pid `child`.

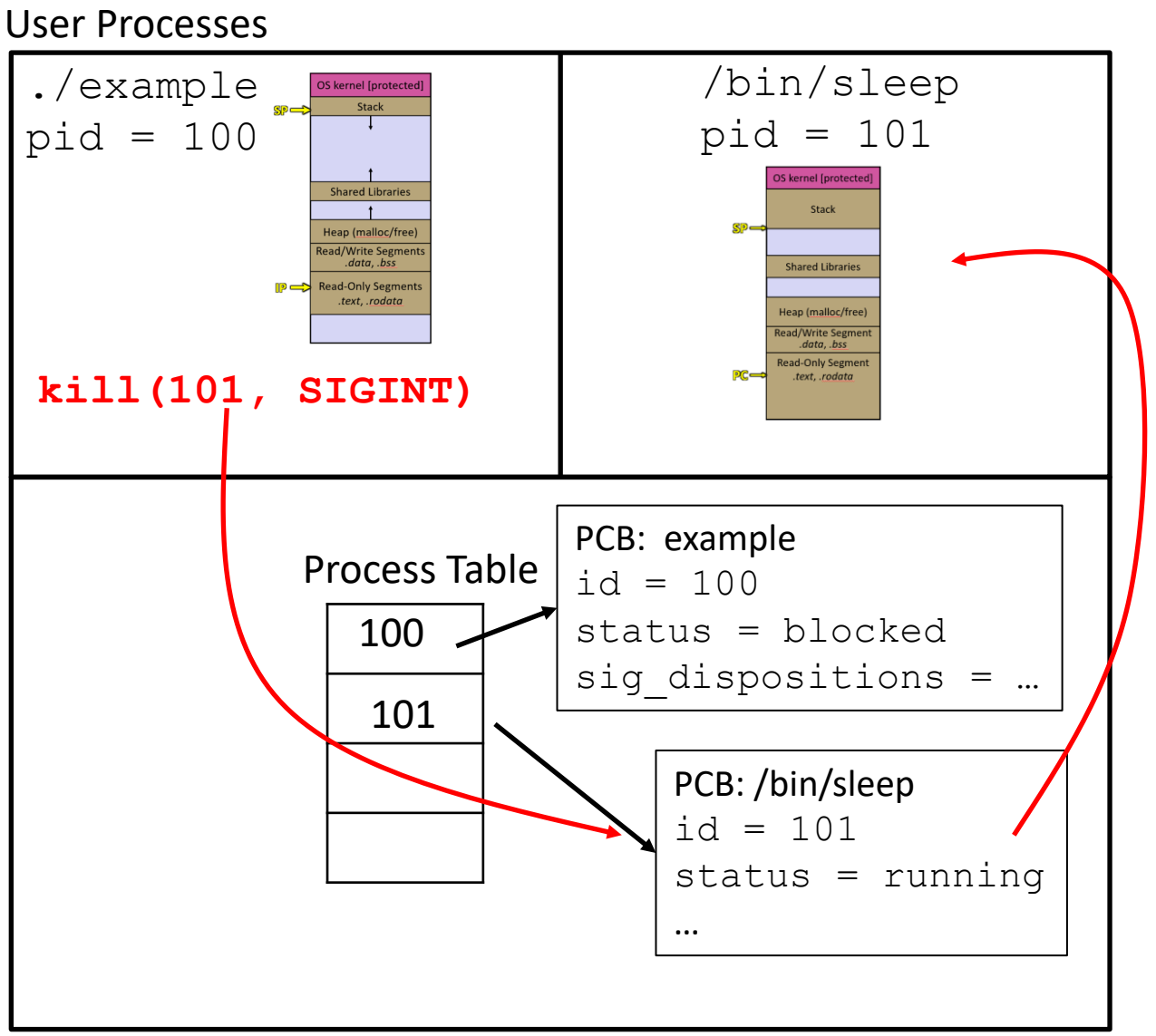
- Eventually, we'll see how `kill` can be used to **send signals to multiple processes at a time**.

Diagram: signals between processes



- './example' is attempting to send a signal to process with pid 101…
  - **Why is it blocked?**
- During a system call, we ask the OS to complete a task for us, so we can not make progress until that is done.

Diagram: signals between processes



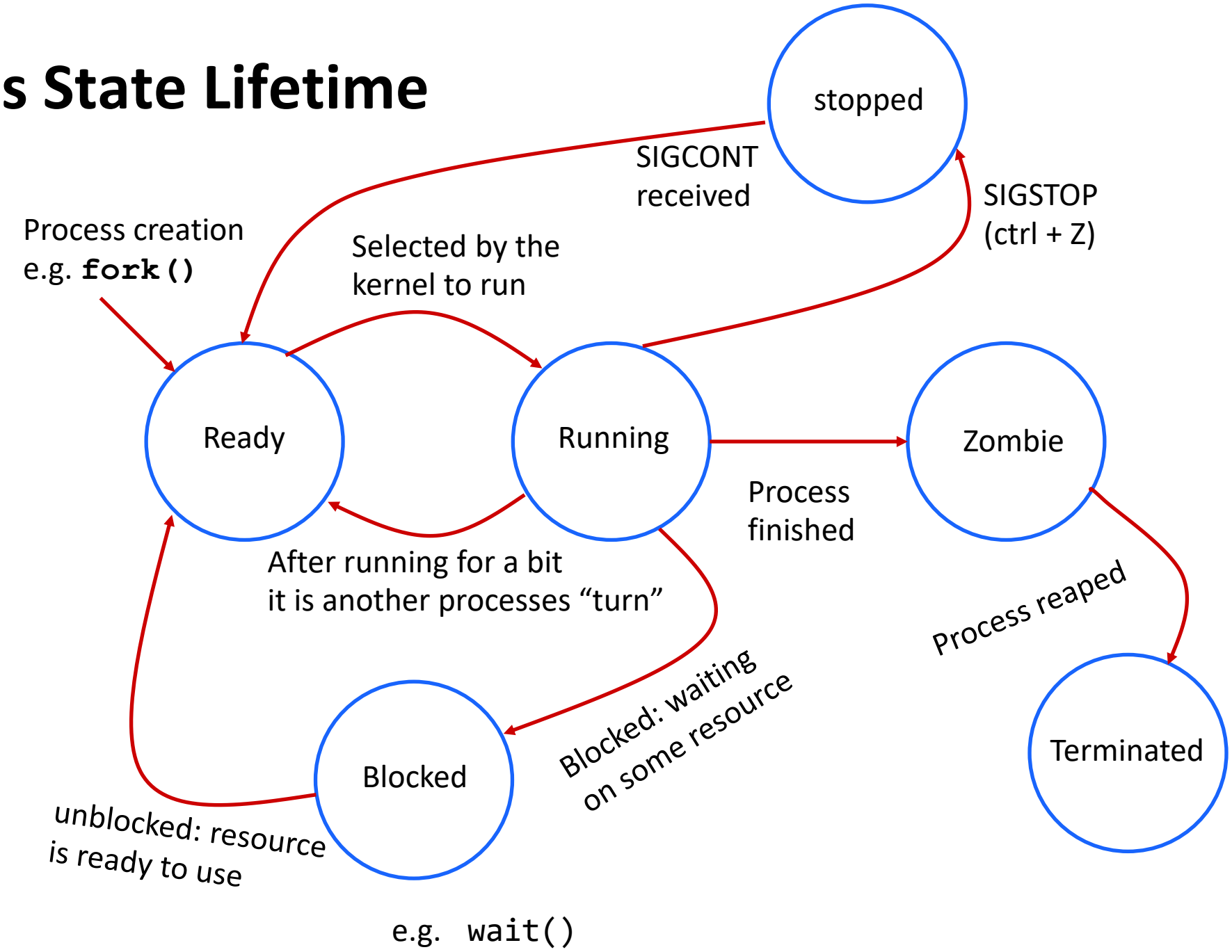
- Signals are sent from process to process via the Operating System.
- This ensures security and enforces that processes only send signals to those they have permission to send to.
  - *Would be weird if **Discord** was allowed to send a SIGKILL to **Chrome**.*
- The OS is here to keep us safe, even from ourselves.

Stopped Jobs

- Processes can be in a state slightly different than being blocked. *// This is relevant for* `penn-shell`
  - When a process gets the signal `SIGSTOP`, the process will not run on the CPU until it is resumed by the `SIGCONT` signal

- Demo:
  - In terminal: `ping google.com`
  - Hit `CTRL + Z` to stop
  - Command: "`jobs`" to see that it is still there, just stopped
  - Can type either "`%<job_num>`" or "`fg`" to resume it

Process State Lifetime



Lecture Outline

- Software signals (resumed from Tuesday)
- **Signal blocking & Signal sets**
- Busy waiting, sigsuspend & sigwait
- Critical sections & signal safety
- wait(), waitpid(), WNOHANG

Previously: Execution Blocking

- When a process calls **`wait()/waitpid()`** and there is a process to wait on, the calling process **blocks.**
- If a process **blocks** or is **blocking** it is not scheduled for execution.
  - It is not run until some condition "unblocks" it
  - For **`wait()`**, it unblocks once the child has transitioned to the "terminated" state.

- This happens frequently when a system call is made, that calling process will block until the system call is completed.

- This is NOT the same as blocking the reception of Signals!
  - Even If if a process is blocked, it can still 'receive' signals…

Signal Blocking

- A process maintains a set of signals called a "signal mask"
  - Signals in that set/mask are "**blocked**"
  - **Signals that are "blocked" are delayed in being delivered to the process**, once unblocked, the process responds to the signals accordingly according to the corresponding disposition.
  - Signals are added to a "pending set" of signals to be delivered once unblocked.

- ***This is not the same as ignoring a signal.***

```
struct sigaction sa = {0};
sa.sa_handler = SIG_IGN;
sa.sa_flags = SA_RESTART;
sigaction(SIGNAL, &sa, NULL);
```

When you set a signal's disposition to SIG_IGN, then when a process receives the signal it simply throws it away.

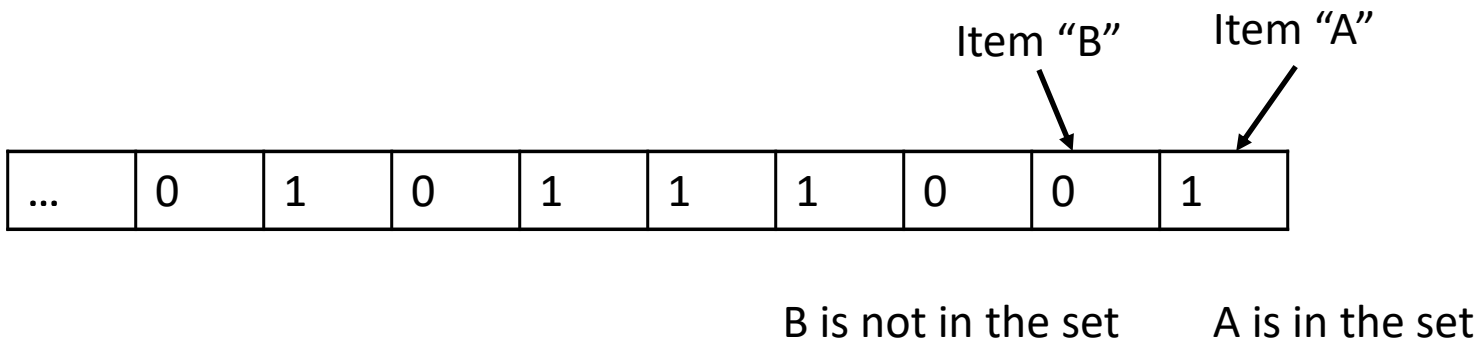
- **Reminder: Process Blocked != Signals are Blocked**

Aside: a way to implement a set in C

- If we have a fixed number of items that can possibly be in the set, then we can use a **bitset**

- Have at least N bits, each item corresponding to a single bit.
  - Each items assigned bit can either be a 0 or a 1, 0 to indicate absence in the set, 1 to indicate presence in the set

- Example:



pollev.com/tqm

- If we have 39 signals, how many bits do we need to have a bitset to represent all signals?
- How many bytes?

sigset_t

sigset_t types must be initialized by a call to **sigemptyset()** when used with a number of different sigsetops. IF NOT THE BEHAVIOR IS UNDEFINED. ☺

- `sigset_t` is a typedef'd bitset to maintain the set of signals blocked
- `int sigemptyset(sigset_t* set);`
  - initializes a sigset_t to be empty
- `int sigaddset(sigset_t* set, int signum);`
  - Adds a signal to the specified signal set
- More functions & details in man pages
  - (`man sigemptyset`)
- Example snippet:

```
sigset_t mask;
if (sigemptyset(&mask) == -1) {
  // error
}
if (sigaddset(&mask, SIGINT) == -1) {
  // error
}
```

sigprocmask()

```
int sigprocmask(int how, const sigset_t* set, sigset_t* oldset);
```

- Sets the process mask to be the specified process “block” mask
- `int` how
  - `SIG_BLOCK`
    - The new mask is the union of the current mask and the specified set.
  - `SIG_UNBLOCK`
    - The new mask is the intersection of the current mask and the complement of the specified set.
  - `SIG_SETMASK`
    - The current mask is replaced by the specified set.

sigprocmask() example

- Example: how do we block SIGINT with sigprocmask?

```
sigset_t mask;
sigset_t old_mask;
sigemptyset(&mask)
sigaddset(&mask, SIGINT)
sigprocmask(SIG_BLOCK, &mask, &old_mask)
```

sigprocmask()

```
int sigprocmask(int how, const sigset_t* set, sigset_t* oldset);
```

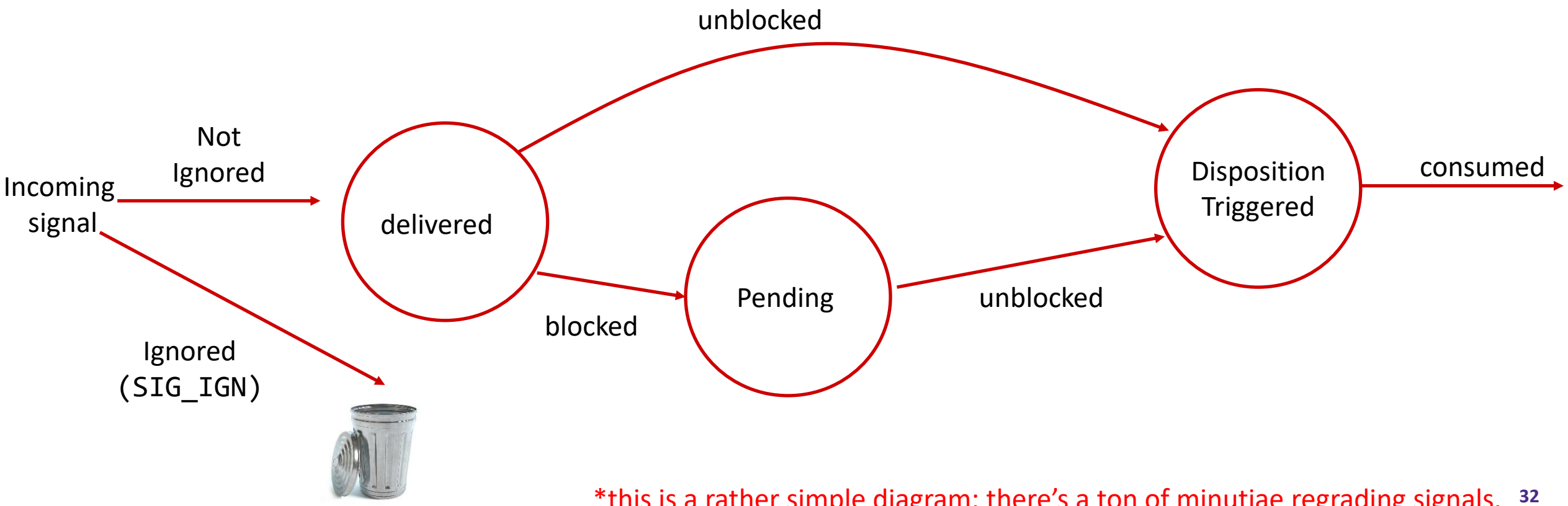
- Sets the process mask to be the specified process mask, set, depending on the value of `int` how
  - "how would you like me to use set?"
- `const sigset_t* set`
  - Is the `set` you would like you use with how
- `sigset_t* oldset`
  - Is set to the previous value of the signal mask.

Demo: `delay_sigint.c`

- Demo: `delay_sigint.c`
  - Installs a custom signal handler for both SIGINT & SIGALRM to know that they were received!
  - Blocks SIGINT (CTRL-C) for the first 5 seconds of the program.
  - Unblocks SIGINT after 5 seconds…
  - CTRL-C should now be able to terminate the program.

Signals are Consumed

- Why didn't the SIGINT kill the program once we restored its disposition?
- When signals trigger their handlers, **they are consumed.**



*this is a rather simple diagram: there's a ton of minutiae regrading signals.

Lecture Outline

- Software signals (resumed from Tuesday)
- Signal blocking & Signal sets
- **Busy waiting, sigsuspend & sigwait**
- Critical sections & signal safety
- wait(), waitpid(), WNOHANG

Busy Waiting

- **Busy Waiting**: when code 'repeatedly' checks some condition, waiting for the condition to be satisfied.
  - Sometimes called *Spinning*, like the phrase "spinning your wheels"
  - This consumes CPU resources while there might be other more meaningful work ready to be scheduled.
  - If we *block*, ***then can we allow another process to make progress while we `wait`…***

- We just did this before, see `delay_sigint.c`

- Demo: running `delay_sigint` and using the terminal command `top` to see the CPU utilization

sigsuspend()

```
int sigsuspend(const sigset_t* mask);
```

- ***Temporarily*** replaces process mask with specified one and suspends execution until a signal that is not blocked is received
- If signal that is not blocked is received, the process *'returns'* from **sigsuspend**
  - The signal first triggers its' handler. Then the mask in place before the suspend call is restored.
  - If the signal received terminates the program, then the process never 'returns' from **sigsuspend.**

- Instead of busy waiting and wasting CPU cycles (that can be used by other processes), we can suspend process execution instead.
- Demo: `suspend_sigint.c`
  - Compare to previous code: `delay_sigint.c`
  - Less CPU resources used ☺

sigwait()

```
int sigwait(const sigset_t *mask, int *sig);
```

- Temporarily suspends execution until a signal in mask **becomes pending**.
  - **IT DOES NOT INSTALL A MASK!**
- Once a signal in the mask becomes pending, **it removes it from the pending set** and returns it in the `int *sig`, parameter.

- For a signal to be pending, you would need to block it using **`sigprocmask`**
- Demo: `sigwait_sigint.c`
  - …be amazed

Blocking

- Calls to `sigsuspend()/sigwait()` **block** until there is information available about a child process (unless you use WNOHANG)…
- Do we always want to block?
  - In the simple cases, yes
    - If the process can not continue because of a shared resource or dependency, then we should block…
  - In more complex cases (like in penn-shell), it may not be desirable…

- A process can make progress on other tasks if we do not block!
  - If we had blocked, those other tasks are also waiting on that task
  - More on this later in the semester when we talk about threads
  - This idea is related to asynchronous programming

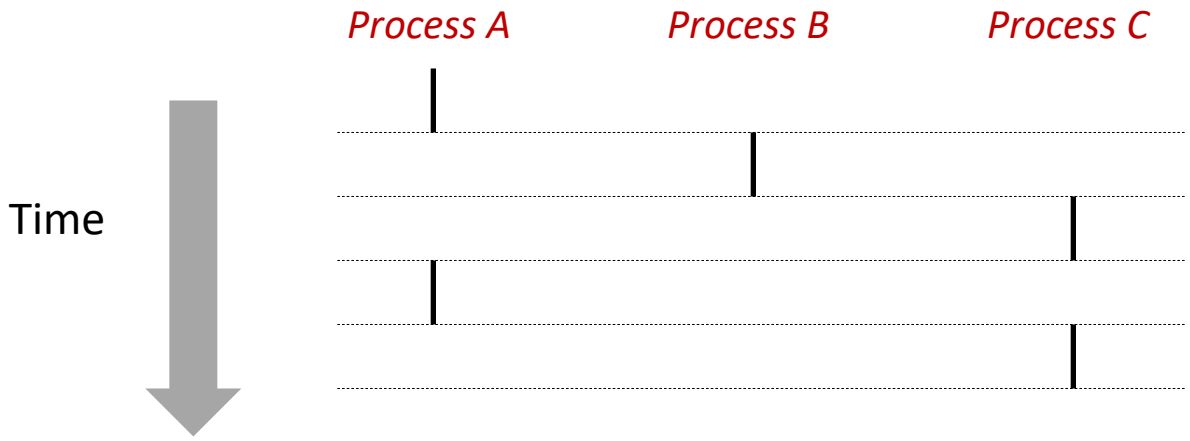
Lecture Outline

- Software signals (resumed from Tuesday)
- Signal blocking & Signal sets
- Busy waiting, sigsuspend & sigwait
- **Critical sections & signal safety**
- wait(), waitpid(), WNOHANG

Concurrent Processes

- Each process is a logical control flow.
- Two processes *run concurrently* if their execution is *interleaved*
- Processes are *sequential* if one is not run until the other is finished.
- Examples running on **single core**:
  - Concurrent: A & B, A & C
  - Sequential: B & C

Note how at any specific moment in time only one process is running

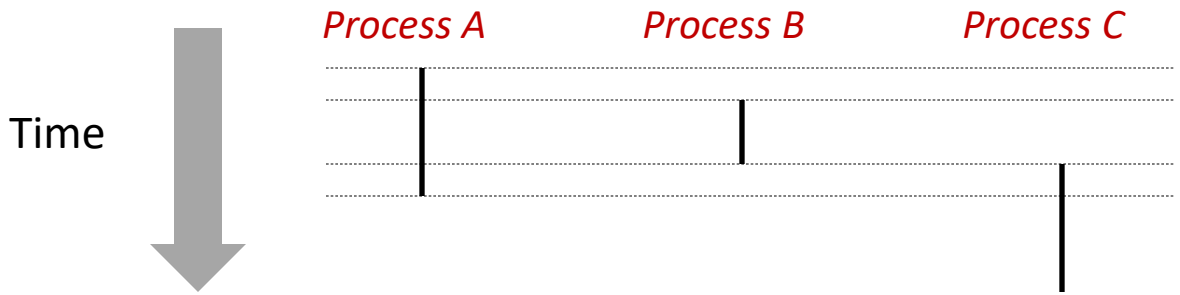


Black line indicates that the process is running during that time

User View of Concurrent Processes

- Control flows for concurrent processes are physically disjoint in time

- However, we can think of concurrent processes as running in parallel with each other

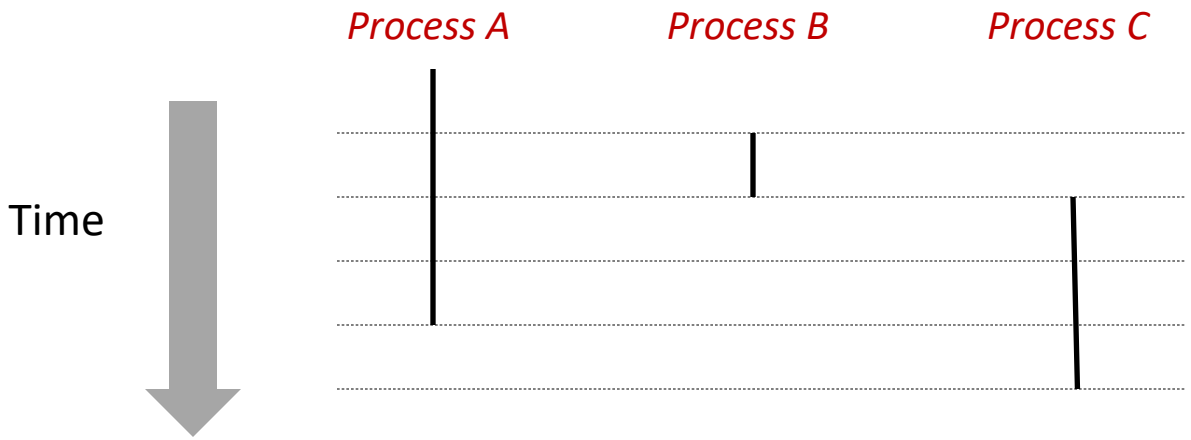


- Above is what a User may **think** is going on. At any moment in time only **one** process has its instructions being executed at a time (ignoring multiple cores).

Parallel Processes

Assuming more than one CPU/CORE

- Each process is a logical control flow.
- Two processes run parallel if their flows overlap at a specific point in time. (Multiple instructions are performed on the CPU at the same time
- Examples (running on dual core):
  - Parallel: A & B, A & C
  - Sequential: B & C



Critical Sections

- There can be issues when one or more resources are accessed concurrently that causes the program to be put in an unexpected, invalid, or error state.

- These sections of code where these accesses happen, called ***critical sections***, need concurrent accesses to be coordinated. Otherwise, the thing being accessed can be put into an invalid state.
  - Think about how in real life we coordinate sharing a water fountain or most things in society.

- Processes are isolated from each other's memory, they only share the OS & file system. The OS & Filesystem protect concurrent accesses for us (mostly)
- **Memory is not protected by concurrent access.** But processes don't share memory… but signal handlers can execute at any time, leading to possible concurrent access of memory within a single process.

It gets worse: Signals Can Interrupt Other Signals

- See code demo: `interrupt.c`
  - Handler registered for SIGALRM and SIGINT
  - Once SIGALRM goes off, it continuously loops and prints
  - SIGINT can be input and run its handler even if SIGALRM was running its handler

Poll Everywhere

pollev.com/tqm

```
// assume this works
void list_push(list *this, float to_push){
    Node *node = malloc(sizeof(Node));
    if (node == NULL)
     exit(EXIT_FAILURE);
    node->value = to_push;
    node->next = NULL;
    this->tail->next = node;
    this->tail = node;
}
void handler(int signo){
    list_push(list, 4.48);
}
int main(int argc, char *argv[]){
    //sa setup omitted, handler set, etc.
    sigaction(SIGINT, &sa, NULL);

    float f;
    while (list_size(list) < 20){
     read_float(stdin, &f);
     list_push(list, f);
    }
}
```

This code is broken. It compiles, but it doesn't ***always*** do what we want. Why?

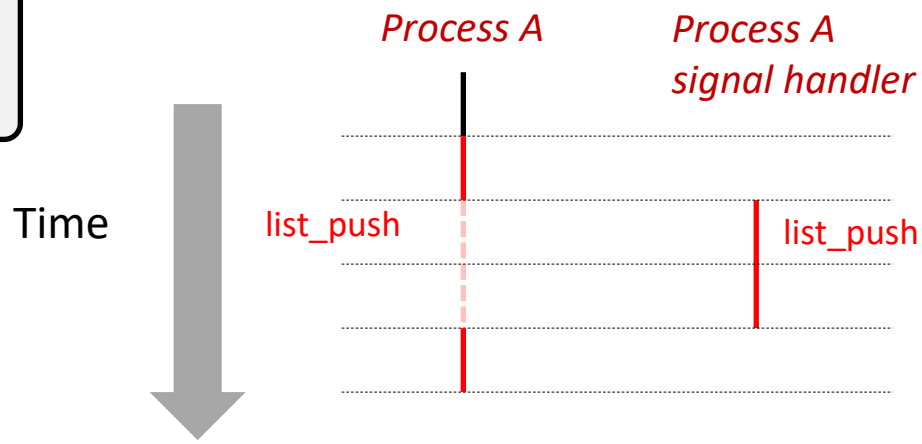
- Assume we have implemented a linked list, and it works
- Assume **`list`** is an initialized global linked list

Critical Section

- There is a critical section in this code!

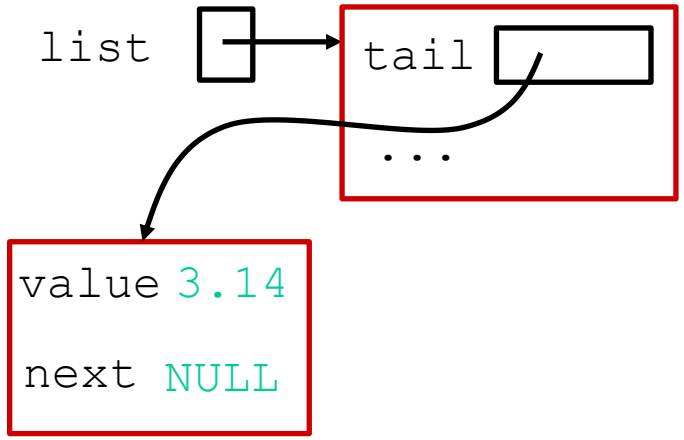
```
void handler(int signo){
    list_push(list, 4.48);
}
int main(int argc, char *argv[]){
    //sa setup omitted, handler set, etc.
    sigaction(SIGINT, &sa, NULL);

    float f;
    while (list_size(list) < 20){
        read_float(stdin, &f);
        list_push(list, f);
    }
}
```

If list_push is interrupted during the pointer rearrangement, we could be left with a malformed linked-list!

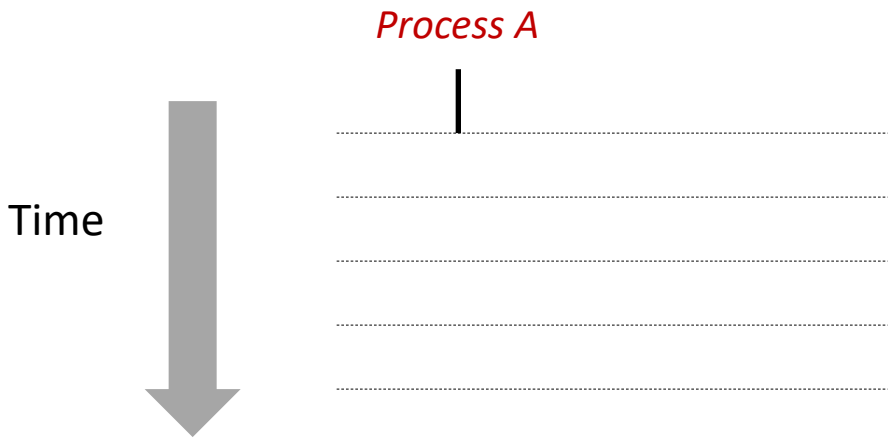


Critical Section Walkthrough

```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```



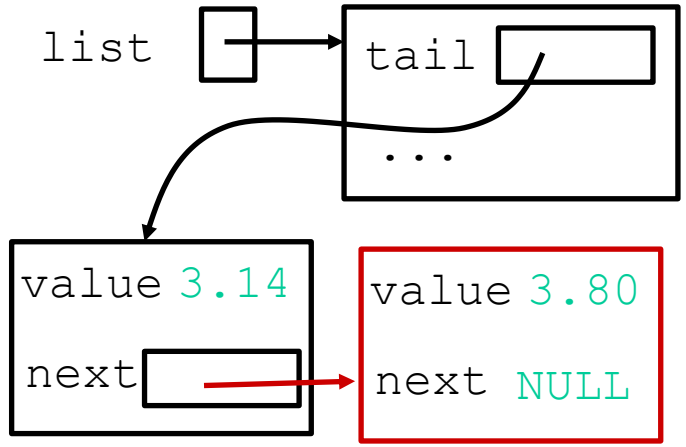
You may assume we also have a “head” pointer that just isn’t shown here.

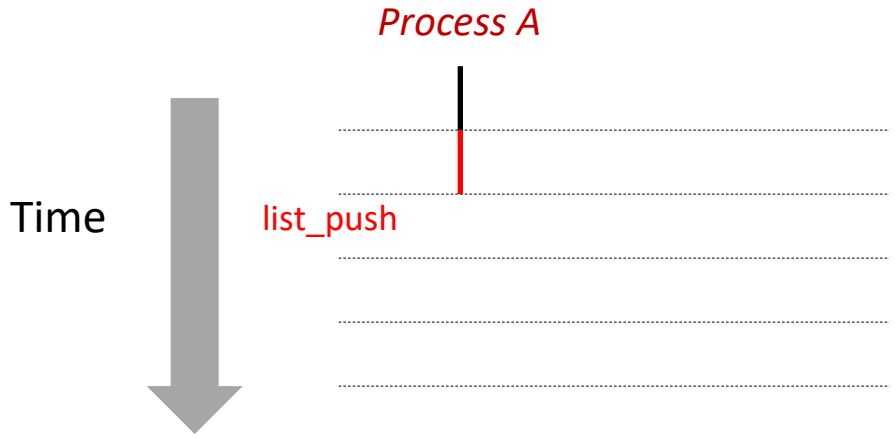


Critical Section Walkthrough

```
//assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node; //to completion
  this->tail = node;
}
```

SIGINT RECEIVED! TIME TO RUN THE HANDLER!

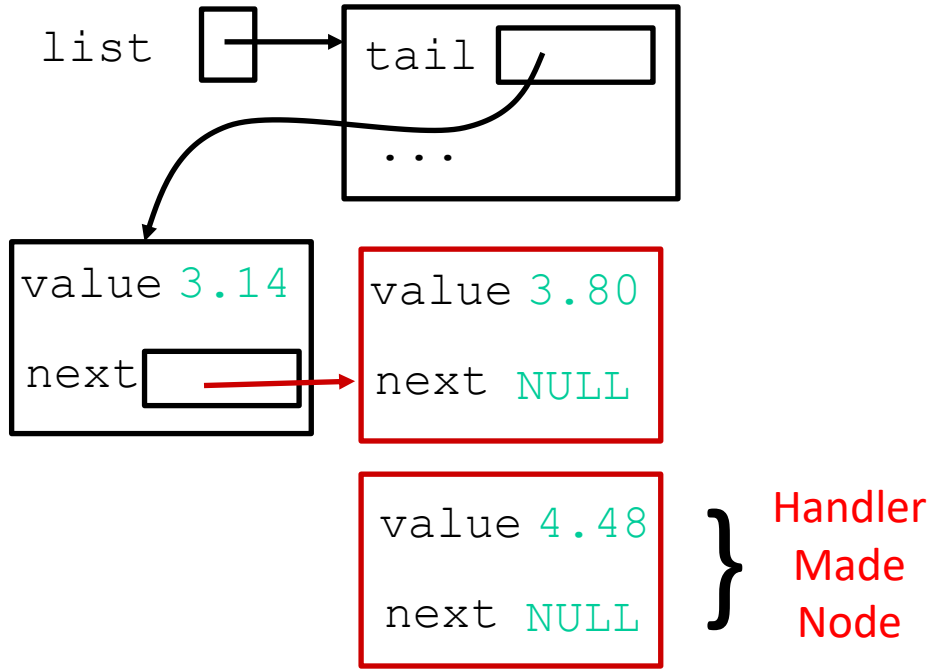




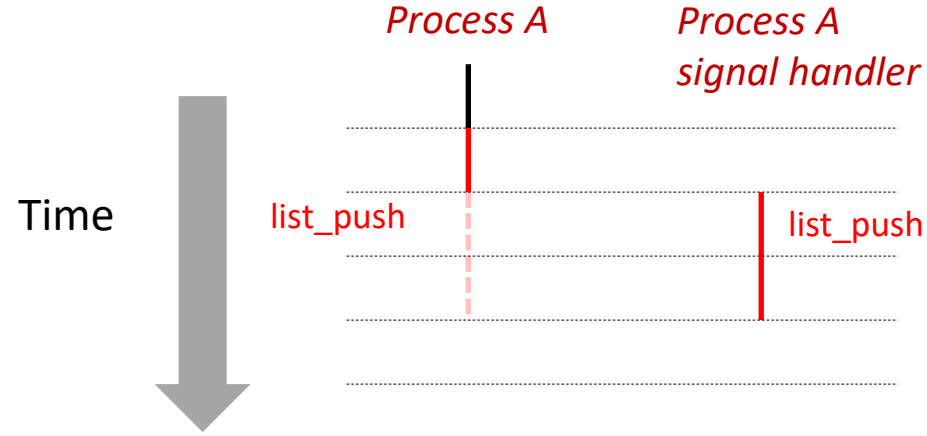
Critical Section Walkthrough

```
//assume this works
void list_
  Node* n
  if (node
    exit(
  }
  node->v
  node->n
  this->t
  this->t
}
```

```
!CALLED FROM THE SIGNAL HANDLER!
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```



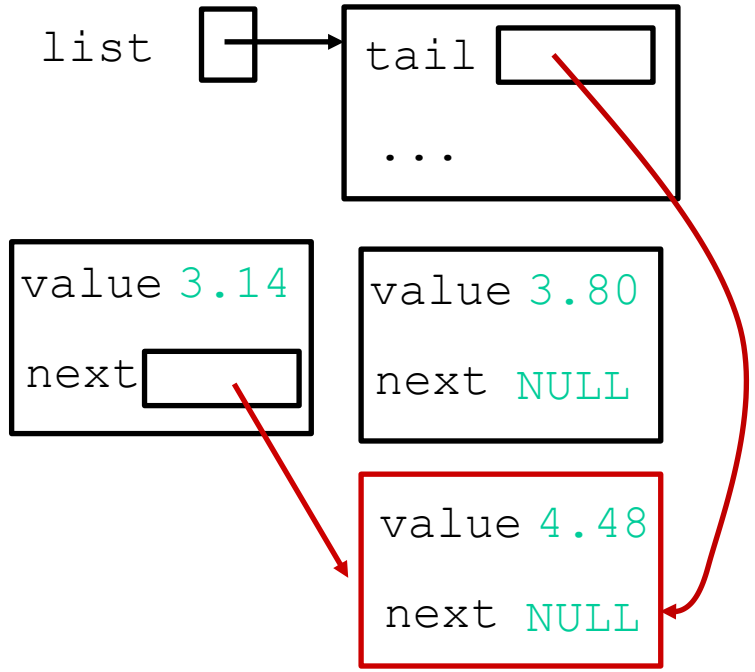
Signal handler interrupts and runs list_push to completion…



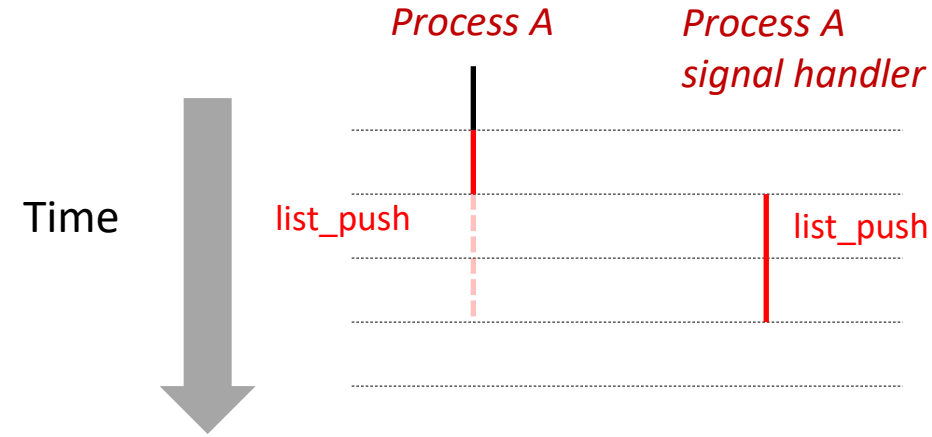
Critical Section Walkthrough

```
//assume this works
void list_
  Node* n
  if (node
    exit(
  }
  node->v
  node->n
  this->t
  this->t
}
```

```
!CALLED FROM THE SIGNAL HANDLER!
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```

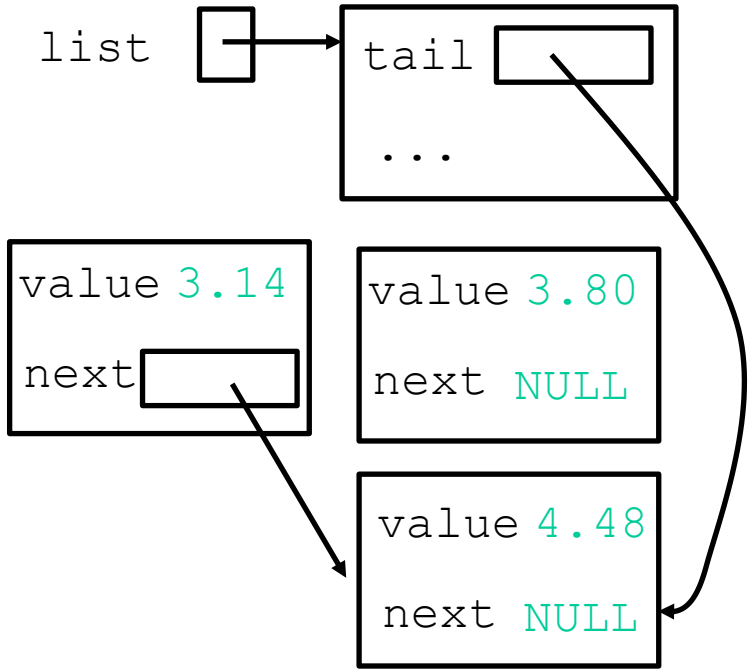


Signal handler interrupts and runs list_push to completion…



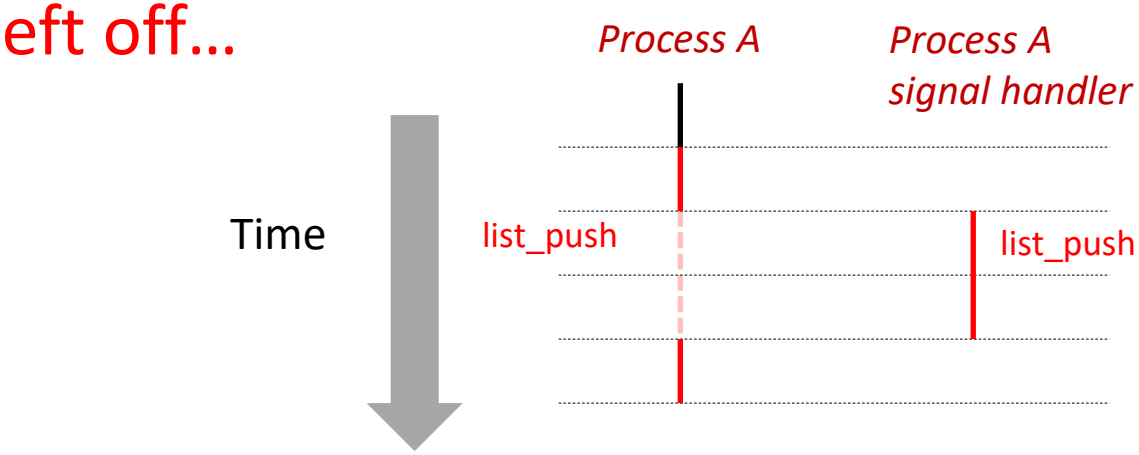
Critical Section Walkthrough

```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node; //our next line to execute.
}
```



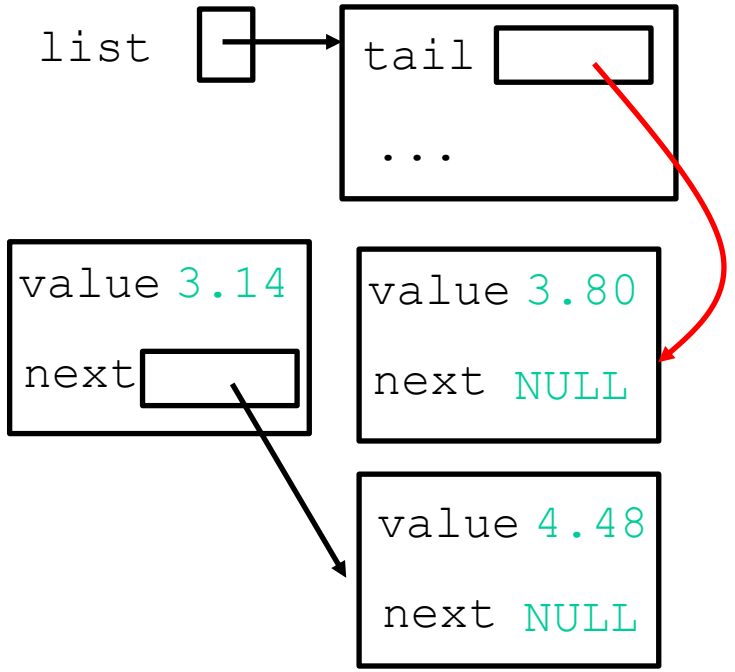
Signal handler finishes…

We return to where we left off…



Critical Section Walkthrough

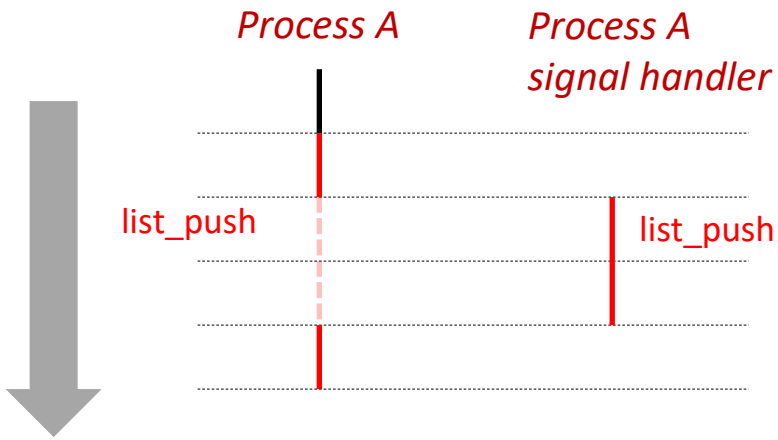
```
// assume this works
void list_push(list* this, float f) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) {
    exit(EXIT_FAILURE);
  }
  node->value = f;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
```



Signal handler finishes…

We return to where we left off…

And we ruined the linked-list.



Poll Everywhere

pollev.com/tqm

```c
// assume this works
void list_push(list* this, float to_push) {
  Node* node = malloc(sizeof(Node));
  if (node == NULL) exit(EXIT_FAILURE);
  node->value = to_push;
  node->next = NULL;
  this->tail->next = node;
  this->tail = node;
}
void handler(int signo) {
  list_push(list, 4.48);
}

int main(int argc, char* argv[]) {
  //signal handler installation
  float f;
  while(list_size(list) < 20) {
    read_float(stdin, &f);
    list_push(list, f);
  }
  // omitted: do stuff with list
}
```

- What can we do to make sure the critical section is safe?
  - *Or, how can we make sure it finished to completion when entered?*

Signal Handlers Block Signals Too

- When a signal handler is triggered, it blocks the signal that triggered it.
- Once the handler is done, it unblocks the signal!

```
struct sigaction {
  void     (*sa_handler)(int);
  int        sa_flags;
  sigset_t   sa_mask;
  ...
};
```

- `sa_mask`
  - Is the mask of signals that are blocked upon the entry of the handler!
  - Even if you set it to empty, it will block the signal that triggered the handler.

Signal Safety

- From **`man 7 signal-safety`**
  - To avoid problems with unsafe functions, there are two possible choices:
    - (a) Ensure that (1) the signal handler calls only async-signal- safe functions, and (2) the signal handler itself is reentrant with respect to global variables in the main program.
      - Prefer this when possible
    - (b) Block signal delivery in the main program when calling functions that are unsafe or operating on global data that is also accessed by the signal handler.
      - Notably: printf, malloc, free, and many functions are not signal safe
      - We can do this with sigprocmask, but (a) is preferred when possible
  - Read more by typing **`man 7 signal-safety`** into the terminal or google

`volatile sig_atomic_t`

- If you need to communicate with a signal handler, we have been using global variables...
  - Modifying global variables is generally unsafe in signals.

- In "real world" code if you want to modify shared data within a signal handler, you should use global variable type: `volatile sig_atomic_t`
  - `volatile sig_atomic_t` is an integer type with interesting properties.
  - Another alternative is writing to a file or pipe. The filesystem is protected by concurrent access for us! (we haven't talked about pipes yet, will get to in a few lectures)

- We will not enforce this in these projects, but we felt like it was worth letting you know.

Lecture Outline

- Software signals (resumed from Tuesday)
- Signal blocking & Signal sets
- Busy waiting, sigsuspend & sigwait
- Critical sections & signal safety
- **wait(), waitpid(), WNOHANG**

Review: `wait()`

- ```
  pid_t wait(int *wstatus);
  ```

  - Calling process waits for ***any*** of its children to exit
    - Also *cleans up* the child process
  - Gets the exit status of child process through output parameter **`wstatus`**
  - Returns process ID of child who was waited for or `-1` on error
  - If you need more nuanced behavior, use **`waitpid()`**
  ```

Review: `waitpid()`

- `pid_t waitpid(pid_t pid, int *wstatus, int options);`

  - `pid` is the pid of the child you would like to *check the status of*
    - When `pid` is set to -1, this is equivalent to waiting for any child
  - **`wstatus`** *tells us how the child has changed*
  - **`options`**, allow us to dictate when waitpid should return
  - Returns process ID of child who triggered the return of **`waitpid`** or `-1` on error.
  - **`waitpid(-1, &wstatus, 0)` is equivalent to `wait(&wstatus)`**

Options for `waitpid()`

- `pid_t waitpid(pid_t pid, int *wstatus, int options);`
  - **`options`**, allow us to dictate when waitpid should return
    - **`WUNTRACED`: waitpid returns if child was stopped**
    - **`WCONTINUED`: waitpid returns if child was continued (via SIGCONT)**
  - Without these options, **`waitpid`** only returns when a child terminates
  - **`options`** can be or'd together
    - `waitpid(-1, &status, WUNTRACED | WCONTINUED);`

Non blocking wait w/ `waitpid()`

```
pid_t waitpid(pid_t pid, int *wstatus, int options);
```

- Can pass in **`WNOHANG`** for **`options`** to make **`waitpid()`** not block or "hang".
- May return
  - process ID of child who triggered the return of **`waitpid`**
  - `-1` on error
  - ***`0` if there are no updates in children processes (specific to `WNOHANG`)***

Quick NOHANG Demo

- Program will fork a child
- make it sleep for 10 seconds
- parent will call waitpid with the NOHANG flag
- then exit…abandoning it's child…
- When a child has been abandoned, **it becomes an orphan…**

quick_nohang.c

`wait/waitpid()` status

- **`status`** output from `wait/waitpid()` can be checked via macro!
- **`WIFEXITED()`** true iff the child exited normally via exit or return from main
- **`WIFSIGNALED()`** true iff the child was terminated via a signal
- **`WIFSTOPPED()`** true iff the child stopped via delivery of signal
- **`WIFCONTINUED()`** true iff the child continued via delivery of signal

Why use `wait()/waitpid`? *CPU Utilization*

- When a process is in a blocked state, it will not be run by the scheduler and thus will not use the CPU

- When analyzing performance, one thing people care about is making maximal use of the CPU. The CPU is what is executing our instructions.
  - Avoiding wasting CPU cycles on things that don't matter
  - Make sure the CPU is running as much instructions (that matter) as possible

Poll Everywhere

pollev.com/tqm

```c
int main() {
  pid_t child_pid = fork();

  if (child_pid < 0) {
    perror("Fork failed\n");
    return EXIT_FAILURE;
  }


  if (child_pid == 0) {
    char* argv[] = {"sleep", "10", nullptr};
    execvp(argv[0], argv);
    return EXIT_FAILURE;
  } else {
    int status;
    pid_t res = waitpid(-1, &status, 0);
    while(res == 0) { /* while no updates*/
      printf("Waiting for child...\n");
      res = waitpid(-1, &status, WNOHANG);
    }
    printf("Done waiting for child!\n");
  }
  return EXIT_SUCCESS;
}
```

- What is the output of this program?
  - *Note: Does it behave as we intend?*

no_hang.c

discuss

```c
int main() {
  pid_t child_pid = fork();

  if (child_pid < 0) {
    perror("Fork failed\n");
    return EXIT_FAILURE;
  }


  if (child_pid == 0) {
    char* argv[] = {"sleep", "10", nullptr};
    execvp(argv[0], argv);
    return EXIT_FAILURE;
  } else {
    int status;
    pid_t res = waitpid(-1, &status, WNOHANG);
    while(res == 0) { /* while no updates*/
      printf("Waiting for child...\n");
      res = waitpid(-1, &status, WNOHANG);
    }
    printf("Done waiting for child!\n");
  }
  return EXIT_SUCCESS;
}
```

- Let's change it to use WNOHANG now.
  - *Is this better?*

A common waitpid bug

```
pid_t child_pid = fork();

if(child_pid == 0){ /* Child Only. */
    sleep(atoi(argv[1]));
    return EXIT_SUCCESS;
}

int res_pid = waitpid(child_pid, &status, WNOHANG);

while(!WIFEXITED(status)){

    if(sleeps < 5) {
        printf("Child has not yet excited. Sleeping for a second; will check after I wake up...\n");
        sleep(1);
        sleeps += 1;

    } else if(!signal_sent){
        printf("It's over for you. You're done.\n");
        kill(child_pid, SIGKILL);
        signal_sent = 1;
    }

    int res_pid = waitpid(child_pid, &status, WNOHANG);
}
printf("I have reaped my child.\nGoodbye.\n");
return EXIT_SUCCESS;
```

Initial State:

```
int sleeps = 0;
int signal_sent = 0;
int status = -1;
```

A common waitpid bug

Discuss

```c
pid_t child_pid = fork();

if(child_pid == 0){ /* Child Only. */
    sleep(atoi(argv[1]));
    return EXIT_SUCCESS;
}

int res_pid = waitpid(child_pid, &status, WNOHANG);

while(!WIFEXITED(status)){

    if(sleeps < 5) {
        printf("Child has not yet excited. Sleeping for a second; will check after I wake up...\n");
        sleep(1);
        sleeps += 1;

    } else if(!signal_sent){
        printf("It's over for you. You're done.\n");
        kill(child_pid, SIGKILL);
        signal_sent = 1;
    }

    int res_pid = waitpid(child_pid, &status, WNOHANG);
}
printf("I have reaped my child.\nGoodbye.\n");
return EXIT_SUCCESS;
```

This doesn't work! It runs forever…
What gives? What might be the issue here?

Initial State:

```c
int sleeps = 0;
int signal_sent = 0;
int status = -1;
```

A common waitpid bug

```
pid_t child_pid = fork();

if(child_pid == 0){ /* Child Only. */
    sleep(atoi(argv[1]));
    return EXIT_SUCCESS;
}

int res_pid = waitpid(child_pid, &status, WNOHANG);

while(!WIFEXITED(status)){

    if(sleeps < 5) {
        printf("Child has not yet excited. Sleeping for a second; will check after I wake up...\n");
        sleep(1);
        sleeps += 1;

    } else if(!signal_sent){
        printf("It's over for you. You're done.\n");
        kill(child_pid, SIGKILL);
        signal_sent = 1;
    }

    int res_pid = waitpid(child_pid, &status, WNOHANG);
}
printf("I have reaped my child.\nGoodbye.\n");
return EXIT_SUCCESS;
```

What gives? What might be the issue here?

Remember: When is WIFEXITED true?

- WIFEXITED is true only when a process terminates normally!
  - (Via Return from Main, or EXIT()…)
- What should we check in addition to this?

A common waitpid bug

```c
pid_t child_pid = fork();

if(child_pid == 0){ /* Child Only. */
    sleep(atoi(argv[1]));
    return EXIT_SUCCESS;
}

int res_pid = waitpid(child_pid, &status, WNOHANG);

while(!WIFEXITED(status) && !WIFSIGNALED(status)){  /* Remember WIFEXITED only exits if the program terminated normally.*/

    if(sleeps < 5) {
        printf("Child has not yet excited. Sleeping for a second; will check after I wake up...\n");
        sleep(1);
        sleeps += 1;

    } else if(!signal_sent){
        printf("It's over for you. You're done.\n");
        kill(child_pid, SIGKILL);
        signal_sent = 1;
    }

    int res_pid = waitpid(child_pid, &status, WNOHANG);
}
printf("I have reaped my child.\nGoodbye.\n");
return EXIT_SUCCESS;
```

This fixes our check, so we exit when either the program has terminated normally or via our sent signal! Neat.

That's all for now!

- Next time:
  - No more signals!!!!!
  - Can move onto file descriptors ☺
- Hopefully you are doing well ☺
- Sign up for a penn-shell partner!